Multiprocessor Scheduling of Precedence-constrained Mixed-Critical Jobs

Dario Socci, Peter Poplavko, Saddek Bensalem, Marius Bozga

Multiprocessor Scheduling of Precedence-constrained Mixed-Critical Jobs

Dario Socci, Peter Poplavko, Saddek Bensalem and Marius Bozga
Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France
{*Dario.Socci* | *Petro.Poplavko* | *Saddek.Bensalem* | *Marius.Bozga*}*@imag.fr*

***Abstract*—The real-time system design targeting multiprocessor platforms leads to two important complications in real-time scheduling. First, to ensure deterministic processing by communicating tasks the scheduling has to consider precedence constraints. The second complication factor is mixed criticality, *i.e.,* integration upon a single platform of various subsystems where some are safety-critical (*e.g.,* car braking system) and the others are not (*e.g.,* car digital radio). Therefore we motivate and study the multiprocessor scheduling problem of a finite set of precedence-related mixed criticality jobs. This problem, to our knowledge, has never been studied if not under very specific assumptions. The main contribution of our work is an algorithm that, given a global fixed-priority assignment for jobs, can modify it in order to improve its schedulability for mixed-criticality setting. Our experiments show an increase of schedulable instances up to a maximum of 25% if compared to classical solutions for this category of scheduling problems.**

I. Introduction

The real-time system design targeting multi and many-core platforms leads to two important issues. Firstly, to ensure deterministic processing by communicating tasks one has to consider scheduling problems with precedence constraints, *i.e., task graphs*. Such tasks often have multiple execution rates and hence their jobs have different arrival times and deadlines [1]. However, the precedence constrained scheduling theory for multiple processors usually considers common arrival times and deadlines of connected jobs. Luckily many practical applications are not sporadic but synchronous-periodic, so they can be modeled by a finite task graph that represents one hyperperiod and enables simple static analysis. We abstract from job periodicity and consider just a static set of jobs with arbitrary statically known arrival times, deadlines, and precedence relations.

Modern technology opens the possibility to integrate upon a single chip various subsystems which required multiple chips and boards in the past, which offers power and weight savings. However, this integration leads to the second issue we raise here – the mixed criticality. The point is that some subsystems are safety critical [2]; therefore, according to current industry standards, one cannot let other subsystems share resources with them, to avoid that their errors and faults have consequences for the safety critical subsystems. The current industry practice assumes complete time or space isolation of subsystems having different levels of criticality, which reduces the benefits of integration. It is much more efficient [3] to let the scheduler use the resources in a flexible way during the normal operation, and only when faults occur give all the resources entirely to the safety critical subsystems, to provide them ample means for fault recovery. In addition, one needs to protect highly critical subsystems from timing misbehavior, especially execution time overruns of less critical ones [4]. For static sets of jobs on single processor, the basic principles and results of corresponding scheduling policies were presented in [3], whereas we investigate extensions towards precedence constraints and multiple processors.

Research supported by the European ICT Collaborative Project no. 288175 (CERTAINTY) and no. 332987 (ARROWHEAD).

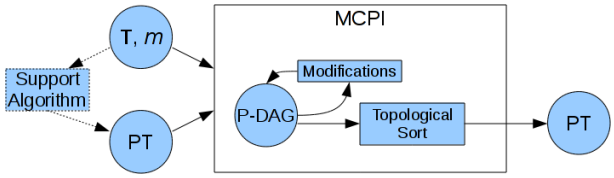


Fig. 1. Proposed algortirhm MCPI. $\mathbf{T}$ stands for task graph and PT for priority table.

For mixed criticality scheduling problems Audsley approach can be used for correct priority assignment [1], but this approach is mainly restricted to uniprocessor scheduling [5]. This is because Audsley approach is based on the assumption that the completion time of the job with the least priority may be computed ignoring the relative priority of the other jobs. This assumption is no longer true in multiprocessors systems. Audsley approach can still be used, by using pessimistic formulas to compute the completion time of the least priority job [5]. However, in the case of finite set of jobs, it can be hard to find a formula with an acceptable level of pessimism. The main contribution of this paper is the *Mixed Criticality Priority Improvement* (MCPI) algorithm, that overcomes the limitation of Audsley approach in multiprocessor system. MCPI, in fact, assigns priorities starting from the highest. This allows us to compute exact completion times. The drawback of this approach is that, unlike Audsley approach, just picking up a job that meets the deadline is not enough for correctness. For this reasons we need an heuristic to help us to select a "good" job in each step. Fig. 1 shows an overview of MCPI. The algorithm takes as input the task graph $\mathbf{T}$, the number of processors m and a priority table PT. The latter may be generated by any known multiprocessor algorithm. We call this algorithm *support algorithm*. The algorithm is based on the concept of *Priority Direct Acyclic Graph* (P-DAG), which defines a partial order on the jobs showing *sufficient* priority constraints needed to obtain a certain schedule. We build such a structure by adding, at each step, jobs from PT, starting from the one with the highest priority. Each time we add a job, we apply a modification to the priority order given by table PT, to

increase the schedulability of safety critical scenarios. When the construction of the P-DAG is terminated, we generate a new priority table by topological sort of the P-DAG.

The paper is organized as follows. Section II-A gives an introduction to the formalism of multiprocessor scheduling in Mixed Critical System. Section III defines P-DAGs and their properties. The MCPI algorithm is then described in Section IV. In Section V we discuss the related work and in Section VI we give experimental results. Finally in Section VII we discuss conclusions and future work.

II. Scheduling Problem

A. Problem Definition

In a dual-criticality Mixed-Critical System (*MCS*), a *job* J_j is characterized by a 5-tuple $J_j = (j, A_j, D_j, \chi_j, C_j)$, where:

- $j \in \mathbb{N}_+$ is a unique index
- $A_j \in \mathbb{Q}$ is the arrival time, $A_j \geq 0$
- $D_j \in \mathbb{Q}$ is the deadline, $D_j \geq A_j$
- $\chi_j \in \{\text{LO}, \text{HI}\}$ is the job's criticality level
- $C_j \in \mathbb{Q}_+^2$ is a vector $(C_j(\text{LO}), C_j(\text{HI}))$ where $C_j(\chi)$ is the WCET at criticality level χ.

We assume that $C_j(\text{LO}) \leq C_j(\text{HI})$[3]. We also assume that the LO jobs are forced to complete after $C_j(\text{LO})$ time units of execution, so $(\chi_j = \text{LO}) \Rightarrow C_j(\text{LO}) = C_j(\text{HI})$. A *task graph* $\mathbf{T}$ of the MC-scheduling problem is the pair $(\mathbf{J}, \rightarrow)$ of a set $\mathbf{J}$ of K jobs with indexes $1 \ldots K$ and a functional precedence relation $\rightarrow \subset \mathbf{J} \times \mathbf{J}$. The criticality of a precedence constraint $J_a \rightarrow J_b$ is HI if $\chi(a) = \chi(b) = \text{HI}$. It is LO otherwise.

A *scenario* of a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$ is a vector of execution times of all jobs: $(c_1, c_2, \ldots, c_K)$. If at least one c_j exceeds $C_j(\text{HI})$, the scenario is called *erroneous*. The *criticality of scenario* $(c_1, c_2, \ldots, c_K)$ is the least critical χ such that $c_j \leq C_j(\chi), \quad \forall j \in [1, K]$. A scenario is *basic* if for each $j = 1, \ldots, K$ either $c_j = C_j(\text{LO})$ or $c_j = C_j(\text{HI})$.

A (preemptive) schedule $\mathcal{S}$ of a given scenario is a mapping from physical time to $\mathbf{J}_\epsilon \times \mathbf{J}_\epsilon \times \ldots \times \mathbf{J}_\epsilon = \mathbf{J}_\epsilon^m$ where $\mathbf{J}_\epsilon = \mathbf{J} \cup \{\epsilon\}$, where ϵ denotes no job and m the number of processors available. Every job should start at time A_j or later and run for no more than c_j time units. A job may be assigned to only one processor at time t, but we assume that job migration is possible to any processor at any time. Also for each precedence constraint $J_a \rightarrow J_b$, job J_b may not run until J_a completes. A job J is said to be *ready* at time t iff:

1) all its predecessors completed execution before t
2) it is already arrived at time t
3) it is not yet completed at time t

The online state of a run-time scheduler at every time instance consists of the set of completed jobs, the set of *ready jobs*, the progress of ready jobs, *i.e.,* for how much each of them has executed so far, and the current criticality mode, χ_{mode}, initialized as $\chi_{mode} = \text{LO}$ and switched to 'HI' as soon as a HI job exceeds $C_j(\text{LO})$. A schedule is *feasible* if the following conditions are met:

Condition 1. *If all jobs run at most for their LO WCET, then both critical (HI) and non-critical (LO) jobs must complete before their deadline, respecting the precedence constraints.*

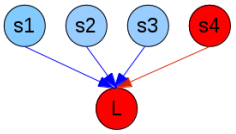


Fig. 2. The task graph of a localization system of an airplane.

Condition 2. *If at least one job runs for more then its LO WCET, than all critical (HI) jobs must complete before their deadline, whereas non-critical (LO) jobs may be even dropped. Also LO precedence constraints are ignored.*

The reason why we allow to have precedences from LO jobs to HI jobs can be seen in the example of Fig. 2. There we have a task graph of the localization system of an airplane, composed of four sensors (jobs s1-s4) and the job L, that computes the position. Data coming from sensor s4 is necessary and sufficient to compute the plane position with a safe precision, thus only s4 and L are marked as HI critical. On the other hand, data from s1, s2 and s3 may improve the precision of the computed position, thus granting the possibility of saving fuel by a better computation of the plane's route. So we do want job L to wait for all the sensors during normal execution, but when the systems switch to HI mode we only wait for data coming from s4.

Based on the online state, a *scheduling policy* deterministically decides which ready jobs are scheduled at every time instant on m processors. A scheduling policy is *correct* for the given task graph $\mathbf{T}$ if for each non-erroneous scenario it generates a feasible schedule. We require that the scheduling policies are *predictable*, *i.e.,* never postponing any jobs when getting less workload.

A task graph $\mathbf{T}$ is *MC-schedulable* if there exists an correct scheduling policy for it. A *fixed-priority* scheduling policy is a policy that can be defined by a priority table PT, which is a vector specifying all jobs in a certain order. The position of a job in PT is its *priority*, the earlier a job is to occur in PT the higher the priority it has. Among all ready jobs, the fixed-priority scheduling policy always selects the m highest-priority jobs in PT. A priority table PT defines a total ordering relationship between the jobs. If job J_1 has higher priority than job J_2 in table PT, we write $J_1 \succ_{PT} J_1$ or simply $J_1 \succ J_2$, if PT is clear from the context. In this paper we assume *global* fixed-priority scheduling which allows non-restricted job migration. A priority table PT is required to be *precedence compliant i.e.,* the following relation should hold:

$$J \rightarrow J' \Rightarrow J \succ_{PT} J' \tag{1}$$

The above requirement is reasonable, since we may not schedule a job before its predecessors complete. The use of fixed-priority in combination with the adopted precedence aware definition of ready job is called in literature *List Scheduling*.

We combine list scheduling with *fixed priority per mode* (FPM), a policy with two tables: PT_{LO} and PT_{HI}. The former includes all jobs. The latter only HI jobs. As long as the current mode is LO, this policy performs the fixed priority scheduling according to PT_{LO}. After the switch to the HI mode, this policy drops all pending LO jobs and applies priority table PT_{HI}. Since scheduling after the mode switch is a single-criticality problem, such a table can be obtained by using classical approaches. Therefore, we focus on producing the table PT_{LO}, in the following simply denoted as PT.

Fixed-priority (FP) online policy (without precedences), is predictable [6], while list scheduling (with precedences) is not. Therefore, online we use a predictable policy described in Sec. IV-B. For predictable policies it is sufficient to restrict the offline schedulability check to simulation of basic scenarios [3]. To be more specific [7], firstly, we check the scenario with execution times $c_j = C_j(\text{LO})$, *i.e.,* the LO scenario. Secondly, for each HI job J_h, we check the scenario where the jobs that completed before J_h have $c_j = C_j(\text{LO})$, while the other jobs (including J_h) have $c_j = C_j(\text{HI})$. Such a scenario is denoted HI$[J_h]$. We check these scenarios offline under list scheduling, and then use their start times as arrival times online.

B. Characterization of Problem Instance

To characterize the performance of scheduling algorithms one uses utilization and related metrics computing the demand-capacity ratio. For a job set $\mathbf{J} = \{J_i\}$ and an assignment of execution times c_i the appropriate metric is load [8]:

$$\ell oad(\mathbf{J}, c) = \max_{0 \le t_1 < t_2} \frac{\sum_{J_i \in \mathbf{J}:\ t_1 \le A_i \wedge D_i \le t_2} c_i}{t_2 - t_1}$$

For a multiprocessor system there does not exist a necessary and sufficient schedulability bound on load, whereas it exists for *uniprocessor systems*: $\ell oad \le 1$. For m-processor system the corresponding bound is only *necessary, but not sufficient* [9]: $\ell oad \le m$. In Section V we discuss also sufficient conditions on load for fixed priority scheduling.

From the problem instance $\mathbf{T}(\mathbf{J}, \to)$ it is convenient to derive the following graphs:

1) HI-criticality task graph: $\mathbf{T}_{\text{HI}}(\mathbf{J}_{\text{HI}}, \to_{\text{HI}})$, where the nodes and edges are the subset of HI jobs and precedences

2) MIX-criticality task graph: $\mathbf{T}_{\text{MIX}}(\mathbf{J}_{\text{MIX}}, \to)$, where the jobs in $\mathbf{J}_{\text{MIX}}$ are obtained from the original set of jobs $\mathbf{J}$ by modifying only job deadlines: $D_{\text{MIX}i} = D_i - (C_i(\text{HI}) - C_i(\text{LO}))$.

For static mixed-criticality jobs, [10] and [11] propose the following characterization of mixed-criticality load:

$$Load_{\text{LO}}(\mathbf{T}) = \ell oad(\mathbf{J}, C(LO))$$
$$Load_{\text{HI}}(\mathbf{T}) = \ell oad(\mathbf{J}_{\text{HI}}, C(HI))$$
$$Load_{\text{MIX}}(\mathbf{T}) = \ell oad(\mathbf{J}_{\text{MIX}}, C(LO))$$

The necessary schedulability condition for load on m identical processors then generalizes to mixed criticality as follows: $Load_{\text{LO}}(\mathbf{T}) \le m \wedge Load_{\text{HI}}(\mathbf{T}) \le m$. However, it was noticed in [11] that in the LO scenario the jobs should meet deadlines $D_{\text{MIX}\,j}$, otherwise deadlines D_j can be missed in a HI scenario, so they made this condition stronger by replacing $Load_{\text{LO}}$ by $Load_{\text{MIX}}$.

Lemma II.1 (Necessary condition for schedulability)**.** *Mixed-critical problem instance* $\mathbf{T}$ *is schedulable only if*

$$Load_{MIX}(\mathbf{T}) \le m \wedge Load_{HI}(\mathbf{T}) \le m \qquad (2)$$

In MIX-criticality graph $\mathbf{T}_{MIX}$ *we should have for all jobs:*

$$A_i + C_i(LO) \le D_{MIX i}$$

whereas in HI-criticality graph $\mathbf{T}_{HI}$ *we should have:*

$$A_i + C_i(HI) \le D_i$$

For practical reasons, we refine the load to a new metric:

$$stress(\mathbf{J}, c) = \max_{0 \le t_1 < t_2} \frac{m}{\min\{m, |\mathbf{J}'|\}} \cdot \frac{\sum_{\mathbf{J}' = J_i | t_1 \le A_i \wedge D_i \le t_2} c_i}{t_2 - t_1}$$

The $m/|\mathbf{J}'|$ scale factor is used to consider the fact that if there are $j < m$ ready jobs then only j processors can be used to schedule them.

Based on *stress*, one can define $Stress_{\text{LO}}$, $Stress_{\text{HI}}$ and $Stress_{\text{MIX}}$. One can also rewrite the necessary conditions (2) using stress, but that would not make them stronger. Nevertheless in general, we have $stress \ge \ell oad$, therefore we use it as a more 'realistic' metric of 'complexity' of the scheduling problem, as for the problem instances of growing complexity it approaches the critical bound m faster than the load.

The formulas of Load and Stress introduced above do not take into account *precedence constraints*. To solve this issue, we define ASAP arrival times and ALAP deadlines, known in the task graph theory [12], but so far mainly used to derive priority tables rather than to compute the load[1].

For a task graph with execution times c, ASAP arrival time A^* is the earliest time when a job can possibly start:

$$A^*_j = \max_i \left(A_j, A^*_i + c_i \ \mid\ J_i \text{ are predecessors of } J_j\right)$$

Dually, ALAP deadline D^* is the latest time when a job is allowed to complete:

$$D^*_j = \min_i \left(D_j, D^*_i - c_i \ \mid\ J_i \text{ are successors of } J_j\right)$$

It is trivial that substituting ASAP arrival time and ALAP deadline to the job parameters does not change the schedulability of the task graph, so the necessary conditions in Lemma II.1 remain valid, whereas the lemma becomes, in general, stronger. It should be noted that, by definition, to compute $Load_{\text{MIX}}$ one should do the ASAP/ALAP calculation in MIX-criticality graph $\mathbf{T}_{\text{MIX}}$ using $C(LO)$, whereas for $Load_{\text{HI}}$ it should be done in graph $\mathbf{T}_{\text{HI}}$ using $C(HI)$. Therefore ASAP arrival and ALAP deadlines for the same job are *mode-dependent*, and one should use these mode-dependent values also for the second part of Lemma II.1, where we check the properties of individual jobs. In the sequel, unless mentioned otherwise, we assume in the algorithms and analysis that the load and stress values are computed using ASAP and ALAP values.

III. Priority DAG

In this section we will introduce the idea of Priority DAG (P-DAG). Informally it is a graph that defines a partial order on the jobs showing *sufficient* priority constraints needed to obtain a certain schedule. This structure makes it easier to reason on priorities than a priority table, since the latter is a total order and thus contains also *non necessary* priority constraints. We will imply for the rest of this section that we are using preemptive list scheduling and we always refer to

[1] In literature the word ALAP is usually used for latest arrival

the basic LO scenario. A priority table PT defines a total order on the set of jobs $\mathbf{J}$ of $\mathbf{T}$. A priority table PT defines one and only one schedule $\mathcal{S}$ when applying list scheduling on m processors, we indicate it with the following notation: $PT \vDash_m \mathcal{S}$.

Consider a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$, a number of processors m and the graph $G = (\mathbf{J}, \rhd)$, where $\rhd$ is a partial order relation defined on $\mathbf{J}$.

Definition 1 (P-DAG)**.** *We call $\mathbf{PT}(G)$ the set of all priority tables that can be obtained by a topological sort of G. G is a P-DAG on m processors for schedule $\mathcal{S}$ iff:*

$$\forall PT,\ PT \in \mathbf{PT}(G) \Rightarrow PT \vDash_m \mathcal{S} \tag{3}$$

Two P-DAGs giving the same schedule are called equivalent.

Definition 2 (Canonical P-DAG)**.** *A* Canonical *P-DAG for a schedule $\mathcal{S}$ is a P-DAG G:*

$$\forall PT,\ PT \in \mathbf{PT}(G) \Leftrightarrow PT \vDash_m \mathcal{S} \tag{4}$$

Let $\mathcal{S}$ be the schedule of a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$ produced by a priority table PT on m processors. Given two jobs J_1 and J_2, we say that J_1 *blocks* by J_2 ($J_1 \vdash_{\mathcal{S}} J_2$) if in the schedule $\mathcal{S}$ there is a point in time t where J_2 is ready but not running while J_1 is running. It's trivial that:

$$J_1 \vdash_{\mathcal{S}} J_2 \Rightarrow J_1 \succ_{PT} J_2 \tag{5}$$

Lemma III.1. *Given a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$, a table PT and a number of processors m. Consider the* blocking *relation $\vdash_{\mathcal{S}}$, where $\mathcal{S}$ is such that $PT \vDash_m \mathcal{S}$. Then $G = (\mathbf{J}, \vdash_{\mathcal{S}})$ is a canonical P-DAG for $\mathcal{S}$.*

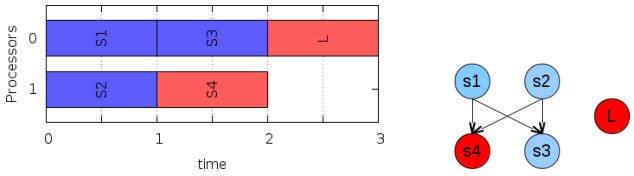
Proof: We need to prove that (4) holds. Let us first prove that G is actually a P-DAG (*i.e.,* (3) holds). This trivially comes from the observation that during the execution of the schedule $\mathcal{S}$, we only need to define a priority when a job blocks another. So the priorities defined by $\vdash_{\mathcal{S}}$ are *sufficient* to generate $\mathcal{S}$.

To prove that the priorities defined by $\vdash_{\mathcal{S}}$ are also *necessary*, let us suppose by contradiction that there exist a table PT' such that $PT' \vDash_m \mathcal{S}$ and $PT' \notin \mathbf{PT}(G)$. The latter means that $\exists\ J_1, J_2$ so that $J_1 \vdash_{\mathcal{S}} J_2$ and $J_1 \not\prec_{PT'} J_2$. By the first statement and by (5), we have $J_1 \prec_{PT'} J_2$ that contradicts the second statement. ■

Example III.1. *Let us consider the tasks of Fig 2, where $\mathbf{J}$ is defined as follows:*

Job	*A*	*D*	χ	*C(LO)*	*C(HI)*
s1	*0*	*3*	*LO*	*1*	*1*
s2	*0*	*3*	*LO*	*1*	*1*
s3	*0*	*3*	*LO*	*1*	*1*
s4	*0*	*4*	*HI*	*1*	*3*
L	*0*	*6*	*HI*	*1*	*3*

consider the priority table $PT = \{s1 \succ s2 \succ s3 \succ s4 \succ L\}$. On two processors PT produces the schedule $\mathcal{S}$ shown in Fig. 3(a). From the figure is easy to derive the blocking relation $\vdash_{\mathcal{S}}$. We have: $s_1 \vdash s_3$, $s_2 \vdash s_3$, $s_1 \vdash s_4$, $s_2 \vdash s_4$. Notice that L is never blocked, because, due to precedence constraints, it is never ready until time 2, when all its predecessors complete. From the blocking relation $\vdash_{\mathcal{S}}$, we can derive the canonical P-DAG $G = (\mathbf{J}, \vdash_{\mathcal{S}})$, shown in Fig. 3(b).



(a) Schedule (b) P-DAG

Fig. 3. The figures of Example III.1.

1: **Algorithm:** *Forest_PDAG*
2: **Input:** **task graph T**
3: **Input:** **priority table** PT
4: **Output:** **P-DAG** G
5: $G = (\emptyset, \emptyset)$
6: **while** $PT \neq \emptyset$ **do**
7: $J^{Curr} \leftarrow PopHighestPriority(PT)$
8: $G.\mathbf{J} \leftarrow G.\mathbf{J} \cup \{J^{Curr}\}$
9: $PT' \leftarrow TopologicalSort(G)$
10: $Simulate(G.\mathbf{J}, PT')$
11: **for** *all trees* $ST \in G$ **do**
12: **if** $\exists\ J' \in ST$: $J' \vdash J^{Curr}$ **then**
13: $G. \rhd \leftarrow G. \rhd \cup \{(J^{Curr}, root(J'))\}$
14: **end if**
15: **end for**
16: **end while**
17: **return** G

Fig. 4. The forest P-DAG generation algorithm

Also, the following is trivial:

Lemma III.2. *If adding an edge to a P-DAG G does not introduce a cycle, the resulting graph G' is still a P-DAG and it is equivalent to G. Also $\mathbf{PT}(G') \subseteq \mathbf{PT}(G)$.*

Definition 3 (Redundant edges)**.** *An edge (J_1, J_2) of a P-DAG G is called* redundant *iff there exists another path in G from J_1 to J_2.*

Removing redundant edges from a P-DAG G will not have any effect on $\mathbf{PT}(G)$. The following trivially follows from Lemmas III.1 and III.2:

Lemma III.3. *Consider a task graph $\mathbf{T} = (\mathbf{J}, \rightarrow)$ and a graph $G = (\mathbf{J}, \rhd)$. Let $\rhd^*$ be the transitive closure of $\rhd$ and $\mathcal{S}$ be a schedule generated by a priority table $PT \in \mathbf{PT}(G)$. Then G is a P-DAG iff:*

$$J' \vdash_{\mathcal{S}} J'' \Rightarrow J' \rhd^* J'',\ \forall J, J' \in \mathbf{J} \tag{6}$$

We are interested in generating P-DAGs that are shaped like forests (*i.e.,* a set of unconnected trees). The reason why we want such a structure will be clear in Section IV, where we use the properties of forest to prove some properties of our algorithm.

We propose in this section an algorithm that generates forest-shaped P-DAG. We will first explain the algorithm and then prove its correctness. The algorithm is shown in Fig. 4, it takes a task graph and a precedence compliant priority table as input and proceeds as follows. The highest priority job J^{Curr} is removed from the table PT and added to the graph G. Then (line 10) we simulate a run of the jobs included in G, using as priority table a topological sort of G. During this sumulation we keep note of the jobs that block J^{Curr} and add an edge

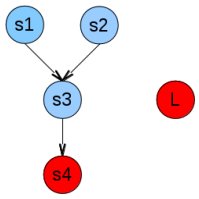


Fig. 5. Forest P-DAG

from J^{Curr} to the root of all the subtrees of G that include a job that blocks J^{Curr}.

Example III.2. *Consider the task graph and the priority table of Example III.1. We will apply Forest_PDAG algorithm to them. In the first step the algorithm picks up s_1, the highest priority jobs from PT, and will add it to the graph. In the second iteration, we pick up s_2, since it is not blocked by any job, we continue without adding any arc. Then we pick up s_3, that is blocked by both s_1 and s_2, so we add the arcs (s_1, s_3) and (s_2, s_3). At the next iteration we pick up job s_4, that is also blocked by both s_1 and s_2, so we add an arc from the root of the tree that contains the blocking jobs (*i.e., *s_3*) to s_4. In the final iteration we pick up job L, that is not blocked by any job, thus we add it to the graph without inserting any arcs from it. The resulting graph is shown in Fig. 5.*

Theorem III.4. *Let G be the graph generated by the* Forest_PDAG *algorithm. Then G is a P-DAG and a forest.*

Proof: We will prove both by induction, by showing that at the n-th step, the statement is true for the partial graph G_n and for priority table PT_n, where both are composed of the first n elements of PT.

Basic step. The basic step is trivial. We have a priority table $PT_1 = \{J_1\}$ with one element and a graph $G_1 = (\{J_1\}, \emptyset)$. A graph of one element is a forest and the only possible topological sort of G_1 gives PT_1.

Inductive step. We know by inductive hypothesis that G_{n-1} is a P-DAG and can generate PT_{n-1}. Also G_{n-1} is a forest. We only add edges from J_n to the root of the trees, this operation may only generate another tree, thus G_n is a forest. Also, since J_n has no parents in G, we can do a topological sort of G_n starting from node J_n, giving it the n-th position on the priority table, same position it has in PT_n At the second step, the partial graph that we have to explore is exactly G_{n-1}, so we can generate PT_{n-1} from it. Since by construction up to the $(n-1)$-th element PT_n and PT_{n-1} are equal, we can generate PT_n by topological sort of G_n.

■

IV. Algorithm

We define here the *Mixed Criticality Priority Improvement* (MCPI) algorithm. It is basically an algorithm to compute offline job priorities under list scheduling, while online we use precedence-unaware global fixed priority with adapted arrival times. As previously discussed, our aim is to overcome the limitation of Audsley approach in multiprocessor systems, by assigning priorities starting from the highest. This allows us to compute exact completion times. We first discuss the offline computation of priorities and then we describe the online policy.

1: **Algorithm:** $MCPI$
2: **Input:** **task graph T**
3: **Input:** **priority table** SPT
4: **Output:** **priority table** PT
5: $SPT \leftarrow PTTransform(SPT)$
6: $CheckLOscenarioSchedulability(\mathbf{T}, SPT)$
7: $G \leftarrow GeneratePDAG(\mathbf{T}, \emptyset, SPT)$
8: $PT \leftarrow TopologicalSort(G)$
9: **if** $anyScenarioFailure(PT, \mathbf{T})$ **then**
10: **return** (FAIL)
11: **end if**

Fig. 6. The MCPI algorithm

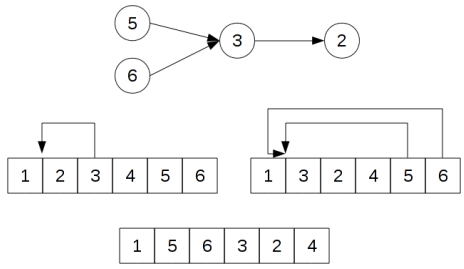


Fig. 7. The initial PT transformation

A. Offline priorities computation

As shown in Fig. 1, MCPI takes as input a priority table, produced by a support algorithm. If we use support algorithm *ALGO*, we indicate that with the following notation: MCPI(*ALGO*). A panoramic of the possible support algorithms is given in Section V.

We use FPM policy, *i.e.,* we have generate two tables, one for LO mode and one for HI mode. As previously discussed, scheduling in HI mode is a single criticality problem. Thus we will compute PT_{HI} for HI with the support algorithm using C(HI) execution times. MCPI is just used to compute PT_{LO}, that we will simply indicate with PT. To build such a PT MCPI takes the priority table generated by the support algorithm and tries to improve the HI scenarios schedulability by increasing the priorities of HI jobs as much as possible without undermining the LO schedulability.

The pseudocode of the algorithm is given in Fig. 6. The algorithm takes as inputs the support priority table SPT and the task graph $\mathbf{T}$. We require SPT to satisfy property (1). In case the support algorithm does not imply property (1), we apply a transformation to SPT such that (1) will hold. The transformation is done as follows. We repeatedly scan the priority table, from the highest to the least. For each job J that has higher priority than some of its predecessors, we rise the priority of those predecessors moving them immediately before J, keeping their relative order. This procedure is illustrated in Fig 7, where we show the task graph, the priority table and its modifications.

We then check LO scenario schedulability. If the schedulability holds, it will be kept as an invariant during the execution. Subroutine $GeneratePriorityDAG$ generates a forest P-DAG, based on the support priority table SPT. It is a modified version of $Forest\_PDAG$. Then we obtain a priority table from G by using the well-known $TopologicalSort$ procedure (see *e.g.,* [13]), which traverses the trees in G from the roots to the leafs while adding the visited nodes to PT. Finally, the subroutine $anyScenarioFailure$ evaluates whether Condition 2

Algorithm: $GeneratePDAG$
Input: **task graph T**
Input: **priority table** SPT
In/out: **P-DAG** G
if $\mathbf{T} \neq \emptyset$ **then**
  $J^{\text{curr}} \leftarrow SelectHighestPriorityJob(\mathbf{J}, SPT)$
  $G.V \leftarrow G.V \cup \{J^{\text{curr}}\}$
  $Simulate(G.\mathbf{J}, PT')$
  for *all trees* $ST \in G$ **do**
    if $\chi(J^{\text{curr}}) = \text{LO}$ **then**
      if $\exists\ J' \in ST\colon J' \vdash J^{\text{curr}} \vee J' \rightarrow J^{\text{curr}}$ **then**
        $InsertAsRoot(J^{\text{curr}}, ST)$
      end if
    else
      $InsertAsRoot(J^{\text{curr}}, ST)$
    end if
  end for
  $PullUp(J^{\text{curr}}, G, SPT)$
  $\mathbf{J} \leftarrow \mathbf{J} \setminus \{J^{\text{curr}}\}$
  $GeneratePDAG(\mathbf{T}, G, SPT)$
end if

Fig. 8. The algorithm for computing priority tree in MCPI

is met. In this case the algorithm succeeds. The check is done by a simulation over the set of all scenarios $HI[J_h]$, as explained in Section II-A.

In Fig. 8 function *GeneratePDAG* is shown. This is a recursive function that takes as inputs the task graph $\mathbf{T}$, the support priority table SPT, and the graph G generated so far (that will be empty at the beginning of the first iteration). The function is very similar to the algorithm of Fig. 4. It selects the highest priority job (*i.e.,* the first unassigned job of table SPT) and adds it to the graph G. Then:

1) if $\chi(J^{\text{curr}}) = LO$*:* we add an arc from J^{curr} to all the root of the trees ST present in G where $\exists\ J'\colon J' \vdash J^{\text{curr}}$. We also add an arc from J^{curr} to the root of the subtrees ST present in G where $\exists\ J'\colon J' \rightarrow J^{\text{curr}}$.

2) if $\chi(J^{\text{curr}}) = HI$*:* an arc from J^{curr} to the root of all the trees present in G is added.

The reason why we add extra arcs, compared to the procedure shown in Fig. 4 is to ensure safety of further modifications of G. These modifications are done by function $PullUp$ when called on J^{curr}. This function is the core of the algorithm. It modifies the P-DAG generated so far trying to improve the HI schedulability of the initial priority order. Notice that if this function were not called, the algorithm would just generate a P-DAG of the initial priority table SPT. After the $PullUp$, we just remove the current jobs from $\mathbf{J}$ and the function is called again recursively.

Function $PullUp$ is described by the pseudocode in Fig. 9. The idea behind this function is to try to improve the schedulability of HI scenarios by raising the priorities of HI jobs, "swapping" their position in the graph with LO jobs while keeping the LO scenario schedulability an invariant.

Function $LOpredecessors(J, G)$ returns the set of direct descendants of LO criticality: $\{J_s \mid J \rhd J_s,\ \chi_s = \text{LO}\}$. At each step in Fig. 9 we pick the least priority predecessor from the working set $PREC$, then subroutine $CanSwap(J, J', G)$ checks if it can be safely swapped. If so, we perform the swap

Algorithm: $PullUp$
Input: **job** J
Input: **priority table** SPT
In/out: **priority tree** G
$PREC = LOpredecessors(J, G)$
while $PREC \neq \emptyset$ **do**
  $J' \leftarrow SelectLeastPriorityJob(PREC, SPT)$
  $PREC \leftarrow PREC \setminus \{J'\}$
  if $CanSwap(J, J', G)$ **then**
    $TreeSwap(J, J', G)$
    $PREC \leftarrow PREC \cup LOpredecessors(J', G)$
  end if
end while

Fig. 9. The pull-up function

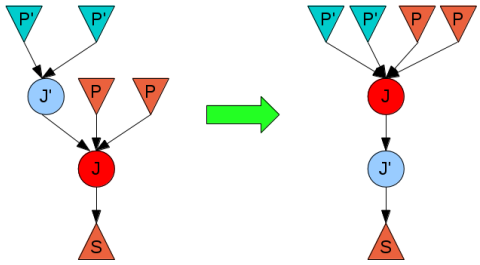


Fig. 10. The effect of a Swap. The red triangle marked with S represent the successors of J, while the triangle marked with P and P' are, respectively the predecessors of J and J'.

and extend the working set of children. The function proceeds until this set is empty. Subroutine $CanSwap(J, J', G)$ performs a 'swap' modification of graph G (described below), thus obtaining new tentative graph G' and a corresponding tentative schedule $\mathcal{S}$. This new schedule is calculated by fixed priority simulation in LO scenario for a $PT' \in \mathbf{PT}(G')$. The new schedule is accepted by $CanSwap$ if it is LO-schedulable. Note that $CanSwap$ immediately rejects to swap J and J' if $J' \rightarrow^* J$, to maintain the precedence compliance of priorities.

Function $TreeSwap(J, J', G)$ performs the following modification on graph G:

1) (J', J) is transformed into (J, J')
2) $if\ \exists J_p\colon (J_p, J'),\ (J_p, J')$ is transformed into (J_p, J)
3) $\forall J_s\colon \exists (J, J_s),\ (J, J_s)$ is transformed into (J', J_s)

The swap is illustrated in Fig. 10. After the swap we update the set $PREC$ to take 3) into account and we reiterate.

Example IV.1. *Consider again the instance and the priority table of Example III.2. Let us apply MCPI on them. The table* PT *is already precedence compliant, so* $PTTransform$ *will not modify it. Then we check LO schedulability, by simulation. The result of the simulation of the LO scenario is the Gantt chart of Fig. 3(a), where it is easy to check that no jobs miss its deadline.*

Then we apply function GeneratePDAG*. In the first iteration we add* $s1$ *to* G*. It is not blocked by any other job, so we proceed with the second iteration.* $s2$ *is added to* G*, again we do not have any blocking. Next we add job* $s3$*, and we have the following blocking relations:* $s1 \vdash s3$ *and* $s2 \vdash s3$*. Thus we add the following edges to* G*:* $s1 \rhd s3$ *and* $s2 \rhd s3$*. Then we add* $s4$*. Since it is a HI job, we add the edge* $s3 \rhd s4$*, since* $s3$ *is the root of the only tree of* G*.*

Since $s4$ *is a HI job, we run* $PullUp$ *on it. First we swap*

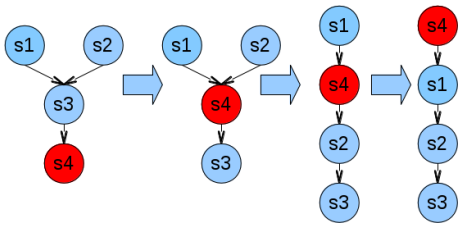


Fig. 11. The effect of function $PullUp$ on job $s4$.

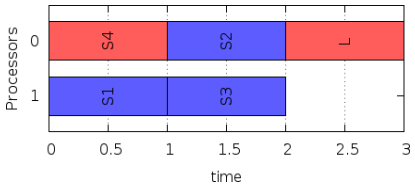
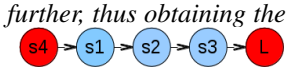


Fig. 12. The schedule obtained by MCPI in Example IV.1.

it with $s3$, after checking that after this operation the jobs will still meet their deadlines. Then we swap it also with $s1$ and $s2$. The result of PullUp *function is shown in Fig. 11. Finally we add job L to the graph and the edge $s3 \triangleright L$. Since $s3 \rightarrow L$, we may not swap further, thus obtaining the following P-DAG:*



From topological sort we obtain the priority table $PT = \{s4 \succ s1 \succ s2 \succ s3 \succ L\}$. The priority table thus obtained leads to the schedule of Fig. 12. The reader may easily verify that using the initial priority assignment, the schedule will fail in scenario HI$[s4]$, where $s4$ will run for 3 times unit, while using the table generated by MCPI the task graph is schedulable in HI$[s4]$ and HI$[L]$.

Theorem IV.1. *The Graph produced by $GeneratePDAG$ procedure is a forest P-DAG.*

Proof: $GeneratePDAG$ proceeds similarly to *Forest_PDAG*, which is correct by Theorem III.4. There are only two differences:

1) it adds more edges at each step
2) it performs the *swap* modification

Since, by Lemma III.2, with extra edges added, G still remains a P-DAG, we observe, by Theorem III.4, that $GeneratePDAG$ ensures that G is a P-DAG at least until the first swap.

To complete the proof we have to show that after the *swap* operation G remains to be a P-DAG. Let us assume by contradiction that, after a swap $TreeSwap(J^s, J^p, G)$, the resulting graph $G' = (\mathbf{T}', \triangleright)$ is no longer a P-DAG. Notice that J^s is a HI job, and thus after inserting it G becomes a connected tree. Also, after the first swap, G is still a tree, such that J^p is the new root and job J^s is the root of a subtree that contains all jobs except J^p (see Fig. 10). After multiple swaps, we will have a tree composed of a chain of LO jobs in the upper part, connected to a subtree that has J^s as root. This is illustrated in Fig. 13.

On the left side of the figure we have a tree with HI job J as root. After swapping J with J',J'' and J''' (in this order), we obtain the tree on the right side. This tree is composed of a chain of J',J'' and J''' and a subtree whose root is J.

By Lemma III.3 and the contradicting hypothesis, we have that G' can generate a table PT' that leads to a schedule $\mathcal{S}$

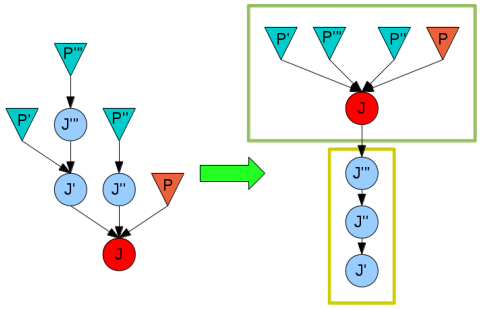


Fig. 13. The effect of multiple Swaps.

such that:

$$\exists J', J'' \colon J' \vdash_{\mathcal{S}} J'' \ \wedge \ J' \not\triangleright^* J''$$

For $TreeSwap(J^s, J^p, G)$ all the possible $J' \vdash J''$ relations that were not present before the swap are such that either $J' = J^p$ or $J^p \rightarrow J'$. This is because, by lowering J^p priority (*i.e.,* shifting forward its execution), it might enter in the execution window of another job. The same holds for its successors in G.

For J^p, we can then rewrite our contradicting hypothesis as follows:

$$\exists J'' \colon J^p \vdash_{\mathcal{S}} J'' \ \wedge \ J^p \not\triangleright^* J''$$

After the swap, J^p is the root of a subtree ST. So $\forall J \in ST, \ J^p \triangleright^* J$. All jobs J'' that are not in ST are in the chain above J^p, this means that $\forall J'', J^p \not\triangleright^* J'' \Rightarrow J'' \triangleright^* J^p$ which implies that $\forall PT' \in \mathbf{PT}(G), J^p \nvdash_{\mathcal{S}} J''$.

Let us now consider jobs J' such that $J^p \rightarrow J'$. An invariant of our algorithm is precedence compliance, *i.e.,* $J^p \rightarrow J' \Rightarrow J' \triangleright^* J^p$. This means that all such J' are in the chain above J^p. The same reasoning as in the previous case holds. ■

Theorem IV.2. *The computational complexity of MCPI is $O(Ek^2 + mk^3 log(k))$, where k is the number of jobs, E the number of edges and m the number of processors.*[2]

B. Predictable Online Policy

The online policy should be predictable, in the sense that lowering the execution times may not increase the termination time. List scheduling is, in general, non-predictable. Therefore, online we execute a predictable policy that behaves the same way as list scheduling in basic scenarios. Recall that offline we check schedulability by simulating all basic scenarios. For each of them we record all jobs start time in a table and provide the table to the online policy. Online, we keep track of the current basic scenario, assuming LO when in LO mode and HI$[J_h]$ when job J_h causes a switch to HI mode. We assume that jobs arrive not at their nominal arrival times, but at their offline start times specified in the table of the current scenario. The modified arrival times ensure that precedences are satisfied. Therefore our online policy uses the default classical global fixed priority scheduling which is known to be predictable.

V. Related Work: Discussion and Analysis

Although our scope is finite set of jobs, most of the literature concerns with instances that have an infinite set of

[2]We will provide all missing proofs in an extended version

jobs, generated by periodic or sporadic tasks. Periodic tasks are said to be synchronous if the offsets between the first arrival of different tasks are statically known. The deadlines can be implicit (*i.e.,* equal to the period), constrained (*i.e.,* less or equal to the period) or pipelined (*i.e.,* larger than period).

Our work can be applied for scheduling the hyperperiod of *periodic synchronous non-pipelined*(*i.e.,* implicit or constrained-deadline) tasks with *precedence constraints*. However, we still consider general real-time policies, even if not originally designed for such systems, as they can be reused as *starting point for our priority-improvement algorithm*. We are particularly interested in the policies tailored for multiprocessor systems, assuming *global fixed priority* for jobs.

1) Multiprocessor Scheduling: Whereas for uniprocessor scheduling a fixed-job-priority algorithm (EDF) is optimal, for multiprocessor case, dynamic job priorities are essential for optimality[5]. Moreover, the EDF heuristic can be very inefficient for multiprocessors. In seminal work of Dhall and Liu [14] it was shown that the *best*, *i.e.,* maximal, load that can be guaranteed for any schedulable job instance for EDF on multiprocessors is no better than for EDF on uniprocessor. For arbitrarily small $\epsilon > 0$ one can find a feasible job instance with load $1+\epsilon$ that is not schedulable by EDF. For this, let us consider m small-deadline jobs with utilization ϵ/m each and one job with utilization 1 and a large deadline. If the last job, which has a large utilization, was given the highest priority then the schedule would be feasible.

In [15] it was shown that in general implicit-deadline periodic task sets under global *fixed priority* for jobs have the following best guaranteed utilization: $(m+1)/2$. Roughly speaking, the fixed priority scheduling can be guaranteed to find a multiprocessor schedule if the system is loaded by no more than *one half*, and even this is only possible if job priorities are well calculated, *e.g.,* the plain EDF cannot provide this guarantee, as explained earlier. Therefore, EDF modifications have been proposed to provide this guarantee. The main idea of several such algorithms is so-called 'separation' of jobs, *i.e.,* separating those that have low and high contribution to load. One of such algorithms is fpEDF, formulated for periodic tasks [15], and later on generalized to sporadic tasks under name *EDF-DS*, where DS stands for *density separation* (see [5] for references). In our notation, this algorithm computes job density as $\delta_i = C_i/(D_i - A_i)$ and it differs from EDF by always giving the jobs with $\delta_i > 1/2$ the highest priority, ties are broken arbitrarily. For the other jobs, the priority is the default EDF. Obviously, this strategy resolves the Dhall-effect counterexample mentioned earlier.

2) Precedence-constrained Scheduling: The *list scheduling* can be seen as generalization of fixed-priority scheduling by handling precedence constraints using *synchronization* between dependent jobs, *i.e.,* including wait for predecessor completion into the condition of job 'ready' status. Synchronization is essential for multiprocessors, whereas for single processor systems it may be sufficient to require precedence compliance of the priority [16], [1]. In both cases, it is generally recognized that the definition of EDF heuristics should be adjusted by using *ALAP deadlines* D^* instead of the nominal deadlines for priority assignment. For example, the list scheduling knows so-called 'ALAP' and b-level heuristics [12]. Single-processor scheduling uses this approach for priority assignment with adjusted deadlines [16]. Sometimes the ALAP-adjusted EDF is a part of an optimal strategy, see [12] for further references.

3) Mixed-critical Scheduling: There are many works on mixed-critical scheduling for uniprocessor systems that assume the FPM scheduling policy and compute priorities either by a variant of Audsley approach or by improving the EDF priorities. Our previous work, MCEDF [7] algorithm can be seen as a combination of the two, also based on P-DAG. However in the present paper we extended the P-DAG analysis to support precedence relation and multiple processors. Moreover we abandon Audsley approach replacing it by more elaborate priority improvement in P-DAGs, while, by the following theorem, offering a generalization of MCEDF.

Theorem V.1. *For single processor and without precedence constraints, MCPI(EDF) is equivalent to MCEDF*[3]

EDF sets the PT in the increasing deadline order, therefore the EDF improvement strategies perform *deadline modification* of HI jobs, reducing their deadlines to improve their priorities *w.r.t.* LO jobs and re-use EDF schedulability analyzes for the modified problem instance. One of the strategies for deadline modifications scales the relative deadline of all HI jobs by the same factor $x, 0 < x < 1$. This strategy was generalized for multiprocessors in [17], where it was combined with EDF-DS.

There are only a few works on precedence-constrained mixed-criticality scheduling. For single processor, [1] generalizes Audsley approach based algorithm OCBP to support precedence constraints for synchronous systems. In [18], multiprocessor list scheduling algorithm was proposed. However, it is restricted to jobs that all have the same arrival and deadline times. Finally, [19] consider pipelined scheduling for task graphs. However, they implicitly assume that the deadlines are large enough, such that they can be ignored during the problem solving, as only period (throughput) constraints were considered and not deadline (latency) ones.

4) Analysis: From the analysis of literature we make the following choices. For multiprocessor scheduling we use density separation, *i.e.,* EDF-DS, for the construction of FPM priority tables: PT_{LO} and PT_{HI} To represent the state-of-the art approach to mixed critical multiprocessor scheduling, we apply deadline modification to the HI jobs, but instead of the deadline scaling, we use the deadlines D_{MIX}, which anyway should be met in the LO mode. In fact, we base the construction of the LO priority table on the MIX-task graph, $\mathbf{T}_{\text{MIX}}$. In this graph we calculate ALAP deadlines. The resulting values for $D_{\text{MIX}}{}^*$ are substituted as the 'deadlines' when calculating the job density and deadline-based priority in the context of EDF-DS. The resulting LO priority table serves as input for MCPI. For fair comparison with related work in the experiments, we use this table as the reference to evaluate the improvement brought by the MCPI into this table. For the HI table PT_{HI} we use the ALAP deadlines calculated in HI task graph $\mathbf{T}_{\text{HI}}$.

VI. Implementation and Experiments

We evaluated the schedulability performance of MCPI comparing it with those the performance of the support algorithms. We randomly generated task graphs with

[3]We will provide all missing proofs in an extended version

m	jobs	arcs	step	δ	σ_s	instances	EDF	EDF-DS	MCPI(EDF)	MCPI(EDF-DS)	diff(%)	diff-DS(%)
2	30	20	0.005	0.01	3.2	128800	20924	21023	27375	27467	**30.83%**	**30.65 %**
4	60	40	0.02	0.05	6	50500	6839	6887	8263	8310	**20.82%**	**20.66 %**
8	120	80	0.05	0.125	12	31575	3065	3082	3521	3538	**14.88%**	**14.80%**

TABLE I. EXPERIMENTAL RESULTS.

integer timing parameters. Every task graph was generated for a target LO and HI stress pair. The method to generate the random problem instances is similar to the one used in [7]. We restricted our experiments to "hard" task graphs, *i.e.,* those satisfying the following formula:

$$Stress_{\text{LO}}(\mathbf{T}) + Stress_{\text{HI}}(\mathbf{T}) \geq \sigma_s \tag{7}$$

The reason of this choice is that task graphs under that line are relatively easy to schedule. We ran multiple job generation experiments, ranging the target of $Stress_{\text{LO}}$ and $Stress_{\text{HI}}$ in the area defined by (7) with a fixed step s. Per each target, ten experiments were run, generating the points lying near the target with a certain tolerance δ. The result of the experiments are shown in Table I. We ran experiments for 2, 4 and 8 processors. For each generated task graph, we checked the schedulability of EDF, EDF-DS, MCPI(EDF), MCPI(EDF-DS). All algorithms were applied using the FPM scheduling policy, the ALAP and ASAP arrivals and deadlines, starting from modified deadline D_{MIX} in the LO mode, as described in Section V. From the result we can see that MCPI gives a big improvement in schedulability compared to the support algorithm, reaching a maximum of 30.83%.

Fig. 14 and Fig. 15 give the contour graph of the density of the generated points in grayscale, where black is the maximum value and white is 0. The horizontal axis is $Load_{\text{LO}}$, the vertical is $Load_{\text{HI}}$. We used $Load$ in the axes because it better reflects required parallelism. Figures from Fig. 14(a) to Fig. 14(d) refer to the experiments made for 2 processors. In particular Fig. 14(a) shows the density of the generated task graphs, Fig. 14(b) shows the percentage of instances schedulable by EDF-DS among the generated ones. Likewise Fig. 14(c) shows the percentage of task graphs schedulable by MCPI (EDF-DS) and Fig. 14(d) shows the percentage of task graphs schedulable by MCPI (EDF-DS) and not schedulable by EDF-DS. As expected the schedulability decreases while the distance from the axis origin increase. Fig. 14(d) is particularly interesting, because it shows how MCPI increases the schedulability over the support algorithm when the load increases. Notice that approximately around point $(1.7, 1.7)$ the density is higher, suggesting that around this point MCPI is more effective.

Figures from Fig. 15(a) to Fig. 15(d) show respectively the same information of figures from Fig. 14(a) to Fig. 14(d), but referred to experiments on 4 processors. From those graph we have confirmation of the conclusions made above. Also in Fig. 15(d) we have an area where MCPI is particularly effective, approximately around point $(3.3, 3.1)$.

VII. CONCLUSIONS

We addressed the problem of multi-processor scheduling of mixed criticality task graphs in synchronous systems. The advantage of our algorithm over state of the art was demonstrated by experiments on a large set of synthetic benchmarks, demonstrating a good improvement in schedulability.

In multi-processor scheduling is hard to apply the Audsley approach, previously proven effective for single-processor mixed-critical scheduling with precedence constraints [1]. Therefore in our algorithm, MCPI , we assign the priorities in a different order. Nevertheless, MCPI still generalizes an Audsley-approach compliant algorithm MCEDF [7], when applied to single-processor instances without precedences.

In future work, we plan to extend the algorithm for multiple criticality levels and to support pipelining.

REFERENCES

[1] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *RTNS'12*, pp. 11–19, ACM, 2012.

[2] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Stanfill, D. Stuart, and R. Urzi, "White paper: A research agenda for mixed-criticality systems," Apr. 2009.

[3] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie, "Scheduling real-time mixed-criticality jobs," *IEEE Trans. Comput.*, vol. 61, pp. 1140 –1152, aug. 2012.

[4] C. Ficek, N. Feiertag, and K. Richter, "Applying the AUTOSAR timing protection to build safe and efficient ISO 26262 mixed-criticality systems," in *ERTSS'2012*, 2012.

[5] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, Oct. 2011.

[6] R. Ha and J. W. S. Liu, "Validating timing constraints in multiprocessor and distributed real-time systems," in *Proc. Int. Conf. Distributed Computing Systems*, pp. 162–171, Jun 1994.

[7] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Mixed critical earliest deadline first," in *Euromicro Conf. on Real-Time Systems*, ECRTS'13, pp. 93–102, IEEE, 2013.

[8] J. W. S. Liu, *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[9] S. Baruah and N. Fisher, "The partitioned multiprocessor scheduling of sporadic task systems," in *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pp. 9 pp.–329, Dec 2005.

[10] H. Li and S. Baruah, "Load-based schedulability analysis of certifiable mixed-criticality systems," in *Intern. Conf. on Embedded Software*, EMSOFT '10, pp. 99–108, ACM, 2010.

[11] T. Park and S. Kim, "Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems," in *Intern. Conf. on Embedded software*, EMSOFT '11, pp. 253–262, ACM, 2011.

[12] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, Dec. 1999.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[14] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

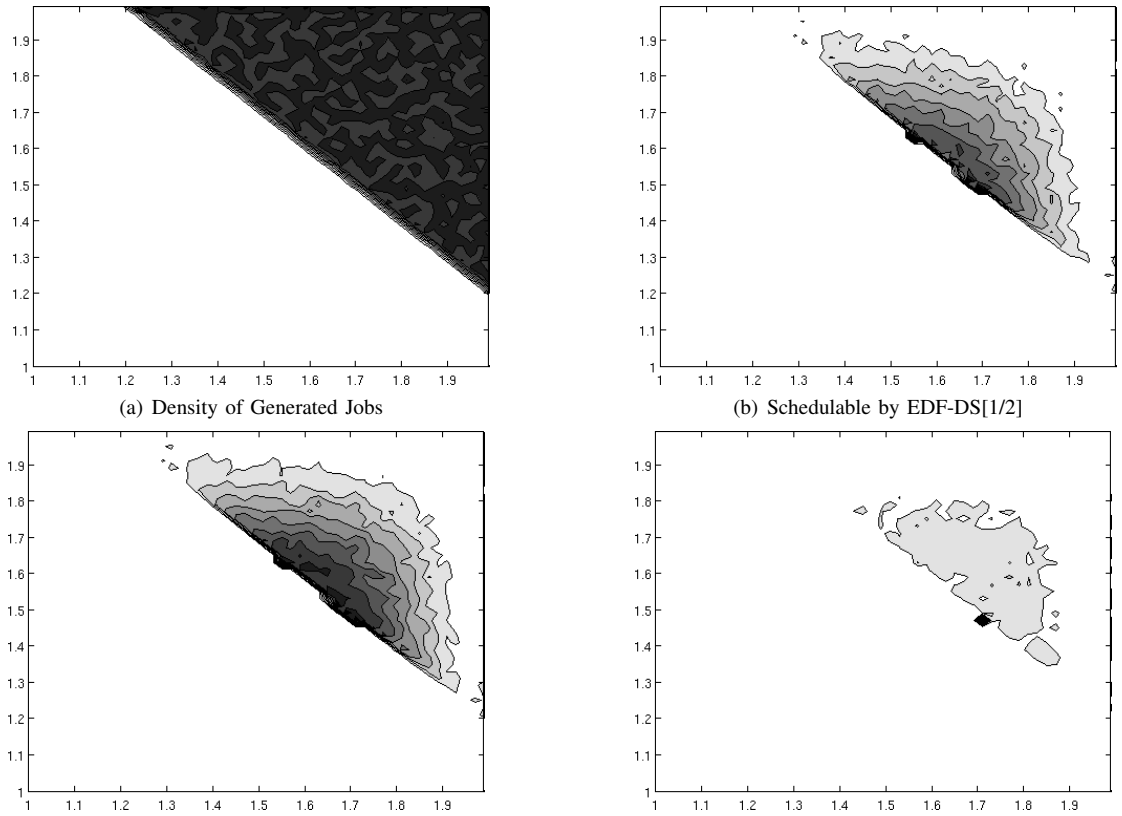
[15] S. K. Baruah, "Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors," *IEEE Trans. Comput.*, vol. 53, pp. 781–784, June 2004.

[16] J. Forget *et al.*, "Scheduling dependent periodic tasks without synchronization mechanisms," in *RTAS'10*, pp. 301–310.

[17] H. Li and S. K. Baruah, "Outstanding paper award: Global mixed-criticality scheduling on multiprocessors," in *24th Euromicro Conference on Real-Time Systems, ECRTS 2012*, 2012.

[18] S. Baruah, "Implementing mixed-criticality synchronous reactive systems upon multiprocessor platforms."

[19] E. Yip, M. Kuo, P. S. Roop, and D. Broman, "Relaxing the Synchronous Approach for Mixed-Criticality Systems," in *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2014.



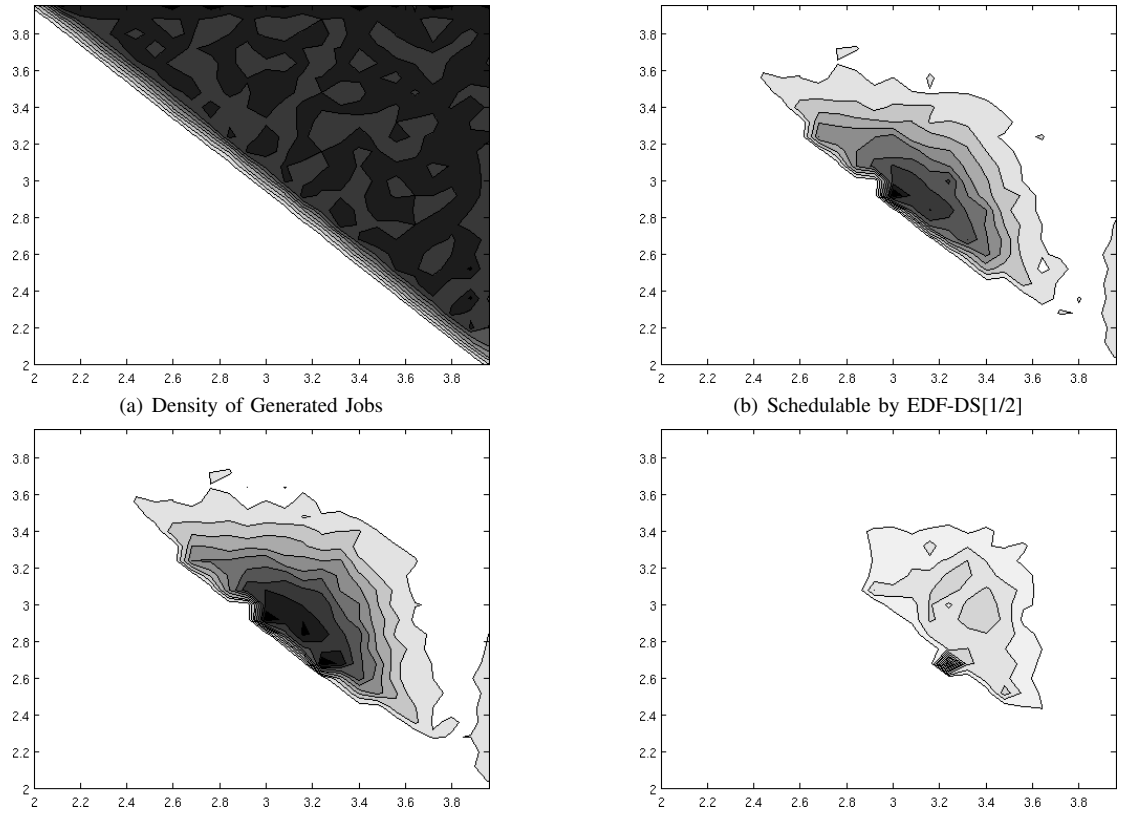
(a) Density of Generated Jobs

(b) Schedulable by EDF-DS[1/2]

(c) Schedulable by MCPI

(d) Schedulable by MCPI and not by EDF-DS[1/2]

Fig. 14. The contour graphs of random task graphs for 2 processors. The horizontal axis is $Load_{\mathrm{LO}}$, the vertical is $Load_{\mathrm{HI}}$.



(a) Density of Generated Jobs

(b) Schedulable by EDF-DS[1/2]

(c) Schedulable by MCPI

(d) Schedulable by MCPI and not by EDF-DS[1/2]

Fig. 15. The contour graphs of random task graphs for 4 processors. The horizontal axis is $Load_{\mathrm{LO}}$, the vertical is $Load_{\mathrm{HI}}$.